ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR ASCON ALGORITHM

SENIOR DESIGN PROJECT

Yunus Emre ERYILMAZ

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JANUARY 2022

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR ASCON ALGORITHM

SENIOR DESIGN PROJECT

Yunus Emre ERYILMAZ
(040180704)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JANUARY 2022

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

RISC-V İŞLEMCİSİNİN KOMUT SETİNİN ASCON ALGORİTMASI İÇİN GENİŞLETİLMESİ

LİSANS BİTİRME TASARIM PROJESİ

Yunus Emre ERYILMAZ
(040180704)

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

OCAK, 2022

We are submitting the Senior Design Project Report entitled as “EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR ASCON ALGORITHM”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Yunus Emre ERYILMAZ
(040180704)

FOREWORD

I would like to thank my mentor Prof. Dr Sıddıka Berna Örs Yalçın who helped me to find this project and who shared her experiences and guided me in every step of the project.

Also, I would like to thank my family and friends for their endless support of my thesis and the other studies.

January 2022 Yunus Emre ERYILMAZ

TABLE OF CONTENTS

Page

ABBREVIATIONS

ALU : Arithmetic Logic Unit
CISC : Complex Instruction Set Computer
EX : Execute
FF : Flip Flop
ID : Instruction Decode
LUT : Lookup Table
LWC : Lightweight Cryptography
NIST : National Institute of Standards and Technology
RAM : Random Access Memory
RFID : Radio-frequency Identification
RISC : Reduced Instruction Set Computer
SAIF : Switching Activity Interchange Format

LIST OF TABLES

Page

LIST OF FIGURES

Page

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR ASCON ALGORITHM

SUMMARY

These days where technology is developing at a rapid pace, the amount of personal data we store in the digital environment increasing. Therefore, data security is important and personal data must be encrypted well. Many asymmetric and symmetric encryption algorithms have been proposed to ensure security but current algorithms are suitable for devices with higher processing power. When these algorithms are applied to constrained devices as RFID tags, smart cards, and low power sensors, the performance of the devices is insufficient. For this reason, the field of lightweight cryptography has emerged to ensure confidentiality in devices with low processing performance and many studies are being conducted on it. One of these studies is the standardization of lightweight cryptography algorithms that were initiated by the National Institute of Standards and Technology (NIST) in 2013. Among the 57 algorithms sent to the study, the ASCON algorithm was chosen in this project due to its speed, simplicity of the operations and small footprint in hardware implementations.

High performance is one of the most anticipated attributes of lightweight cryptology algorithms. One of the ways to improve performance is to design a processor with higher processing power, but this way is very costly and time-consuming. Another way is to expand the instruction sets of an existing processor. This way was aimed at the project. Ibex core, the RISC-V processor designed by lowRISC, was chosen as the processor to expand its instruction set. Then, the C program in ASCON's Github repository was run over Ibex, the energy consumed, the execution time of the program, the size of the program and the area covered by the processor were measured. After that, the most used operation in the program was determined with the SPIKE RISC-V simulator. It was observed that the s-box and rotation operations were mostly used. These two operations are designed in the Verilog hardware description language and included in the arithmetic logic unit of Ibex. In addition, the compiler was modified to call the added commands, and the ASCON program was rewritten with the inline assembly method. Finally, the program was compiled again and the instruction set was run on the modified processor and the energy consumed, the execution time of the program, the size of the program and the area covered by the processor were measured and compared with the previous measurements.

As a result, the blocks we have added have significantly increased the performance of the program, the area covered by the processor has increased, the energy consumed, the program size and the encryption time have decreased.

RISC-V İŞLEMCİSİNİN KOMUT SETİNİN ASCON ALGORİTMASI İÇİN GENİŞLETİLMESİ

ÖZET

Teknolojinin büyük bir hızla geliştiği bu çağda dijital ortamda sakladığımız kişisel veri miktarı artmaktadır. Bu yüzden veri güvenliği büyük önem arz etmekte, veriler iyi bir şekilde gizlenmek zorundadır. Veri güvenliğini sağlamak için birçok asimetrik ve simetrik şifreleme algoritması ortaya atılmıştır ancak güncel algoritmalar işlem gücü daha yüksek cihazlar için uygundur; RFID etiketleri, akıllı kartlar, düşük güçlü sensörler gibi kısıtlı cihazlarda uygulandığında performansı yetersiz kalmakta. Bu nedenle işlem gücü düşük cihazlarda gizliliğin sağlanması için hafif kriptoloji alanı ortaya çıkmış ve hakkında birçok çalışma yapılmaktadır. Bu çalışmalardan biri Ulusal Standartlar ve Teknoloji Enstitüsü (NIST)'nün 2013'te başlattığı standardizasyon çalışmasıdır. Çalışmaya gönderilen 57 algoritma arasından hızı, yapılan işlemlerin basitliği ve donanım gerçeklemelerinde az yer kaplaması sebebiyle bu projede ASCON algoritması seçilmiştir.

Yüksek performans, hafif kriptoloji algoritmalarından en çok beklenen niteliklerden biridir. Performansı arttırmanın yollarından biri daha yüksek işlem gücüne sahip bir işlemci tasarlamaktır ancak bu yol epey maliyetlidir ve zaman gerektirir. Bir diğer yol ise var olan işlemcilerin komut setlerini genişletmektir. Projede de bu hedeflenmiştir. Komut seti genişletilecek işlemci olarak lowRISC'in tasarladığı RISC-V işlemcisi Ibex core seçilmiştir. Ardından ASCON'ın Github reposundaki C programı Ibex üzerinden çalıştırılmış, harcadığı enerji, kaç saat darbesinde şifreleme yapıldığı, programın boyutu ve işlemcinin kapladığı alan ölçülmüştür. Bundan sonra program SPIKE RISC-V simülatörü ile hangi operasyonun en çok kullanıldığı tespit edilmiştir. Simülasyon sonucunda en çok s-box ve döndürme operasyonlarının kullanıldığı gözlenmiştir. Bu iki operasyon Verilog donanım tanımlama dilinde tasarlanıp Ibex'in aritmetik lojik birimine dahil edilmiştir. Bunun yanında eklenen komutları çağırmak için derleyici düzenlenmiş, ASCON programı inline assembly yöntemiyle tekrar yazılmıştır. Son olarak, program tekrar derlenip komut seti genişletilmiş işlemcide çalıştırıldı; harcadığı enerji, kaç saat darbesinde şifreleme yapıldığı, programın boyutu ve işlemcinin kapladığı alan ölçüldü ve daha önceden alınan ölçümlerle karşılaştırıldı.

Sonuç olarak, eklediğimiz bloklar sayesinde programın performansında gözle görülür derecede artış gerçeklemiş, işlemcinin kapladığı alan artmış, harcadığı enerji, program boyutu ve şifreleme süresinde azalma gözlenmiştir.

1. INTRODUCTION

Usage of constrained devices such as RFID (Radio-Frequency Identification) tags, sensors, microcontrollers, and smart cards is increasing every year. This situation brings new security concerns. However, applying cryptographic algorithms is challenging because conventional cryptographic algorithms are not suitable for constrained devices. The algorithms are optimized for systems with high computing power like desktop and server environments. If the conventional algorithms run on constrained devices, they work in low performance. So, lightweight cryptography has emerged [1]. NIST (National Institute of Standards and Technology) started to research standardization of lightweight cryptographic algorithms in 2013 due to the insufficiency of current NIST-approved cryptographic standards. In 2018, they opened a competition for adding a public comment to the standardization process [2]. ASCON is one of the competitor algorithms and finalists. ASCON is chosen for the project because most of its operations can be implemented on FPGA easily due to the behaviour of the operations [3]. Speed is the most important thing about lightweight cryptography. There are several ways to speed up the process. The first of them is the optimization of the algorithm, but it's not feasible and an easy way to do it. Another of them is using a more powerful processor. However, the processor requires a wider area and more power consumption if its computing power gets higher. For this environment, an application-specific processor is a more appropriate way. A RISC-V

based core, Ibex is selected to implement the algorithm because Ibex is a fast and small open-source processor and supports I, M and C extensions[7].

The project aims to increase the performance of the algorithm by extending the instruction set architecture of Ibex core.

1.1 RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) that uses RISC principles. It is configured as a small base ISA with various optional extensions. The basic ISA is very simple, making RISC-V suitable for research and education, yet complete enough to be an ISA suitable for inexpensive, low-power embedded devices [4]. To understand RISC-V ISA, first of all, it is necessary to examine and explain the RISC and CISC principles and the concept of ISA.

1.1.1 RISC and CISC

The Reduced Instruction Set Computer (RISC) is a type of processor architecture with a small, highly optimized instruction set rather than the more specialized sets often found in other types of architecture such as the Complex Instruction Set Computer (CISC). To briefly consider the difference between them, CISC processors may use more than one clock cycle to execute an instruction. On the other hand, in RISC, all commands are finished in 1 cycle. If we consider the multiplication process to better explain this difference, in CISC this process is completed in 1 clock cycle, while in RISC, 4 clock cycles are required to perform this operation.

1.1.2 RISC-V ISA

RISC-V includes 6 different base instruction formats. These are R-type, I-type, S-type, B-type, U-type, and J-type instruction formats. In this notation, rs1 and rs2 are the addresses of the source registers, and rd is the destination register address, which is 5 bits long for a 32-bit base ISA. The instruction formats are shown in Figure 1.1.

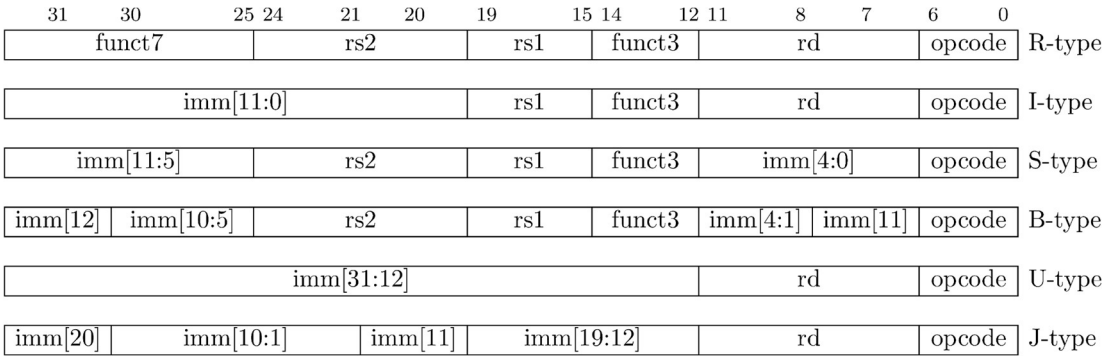


Figure 1.1: Base instruction formats of 32-bit RISC-V ISA[5]

RISC-V ISA contains some special and some general-purpose registers. The naming and usage purposes of these registers are given in Figure 1.2.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 1.2: Registers of RISC-V ISA[6]

x0 (zero) - Hardwired zero: A zero register refers to a special-purpose register that is hardwired to the integer constant 0.

x1 (ra) - Return address: ra holds the return address. This is a memory value in the code region.

x2 (sp) - Stack pointer: The register that holds the address for the stack is called a stack pointer (SP) because its value always points to the top of the stack. Push and pop operations are implemented by decrementing or incrementing the stack pointer.

x3 (gp) - Global pointer: Enables fast access to global (static) data structures. gp is initialized early in the program and never changed.

x4 (tp) - Thread pointer: Enables access to thread-specific data in multi-threaded applications. tp is typically initialized early in the execution of a new thread and never changed.

x8 (s0/fp) - Frame pointer: The frame pointer points to the base of the stack frame.

x5-x7 and x28-x31(t0-t6) - Temporary registers: Holds temporary values that do not persist after function calls.

x9 and x18-x27 (s1-s11) - Saved registers: Holds values that persist after function calls.

x10-x11 (a0-a1) - Function arguments/Return values: Holds the first two arguments to the function or the return values.

x12-17 (a2-a7) - Function arguments: Holds any remaining arguments.

1.2 Ibex Core

Ibex is an open-source 32-bit RISC-V core that supports I, M, and C extensions. It is written in SystemVerilog and consists of parametrizable blocks. Also, it is suitable for embedded control applications[7]. The architecture of Ibex core is shown in Figure 1.3.

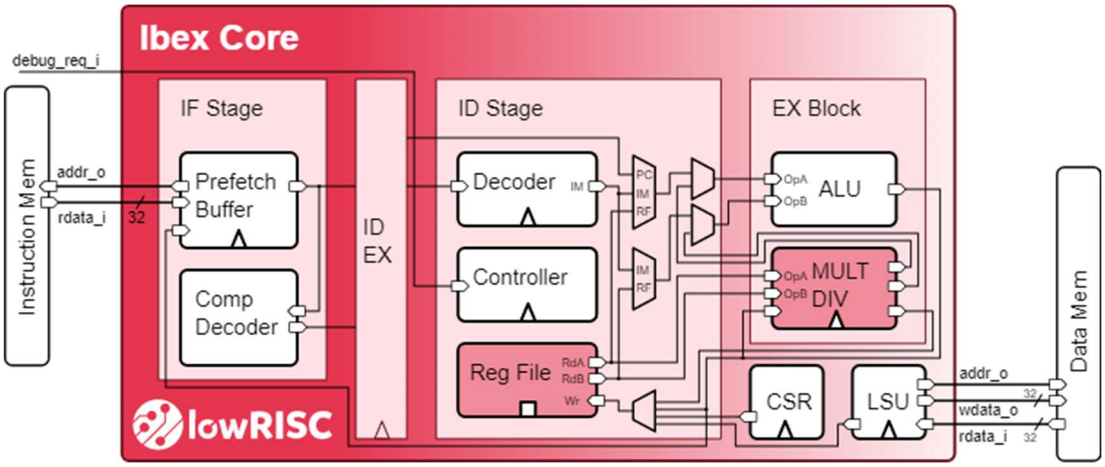


Figure 1.3: Architecture of Ibex with a 2-stage pipeline.

1.3 Ascon Algorithms

ASCON is a family of cryptographic algorithms which consist of authenticated encryption and hashing. It is designed to be lightweight and easy to implement on constraint devices with countermeasures for side-channel attacks. Also, it is selected for the final portfolio of the CAESAR competition, and a finalist competitor in the ongoing NIST Lightweight Cryptography competition. [3]

In authenticated encryption, there are encryption and decryption algorithms. The encryption algorithm gets a maximum 160-bit key (K), 128-bit nonce (N), and associated data (A) of arbitrary length to encrypt plaintext (P) of arbitrary length and produce ciphertext (C) of the same length as plaintext and 128-bit tag (T). Decryption algorithm gets key, nonce, associated data, ciphertext, and tag, which are the same length as that used in encryption, to decrypt the ciphertext and produce plaintext or a verification tag ($\perp$) if the verification of tag fails. Encryption and decryption algorithms can be seen in Figure 1a and 1b, respectively. In Figure 1, k is the length of the key, r is data block size, and a and b are internal round numbers. [3]

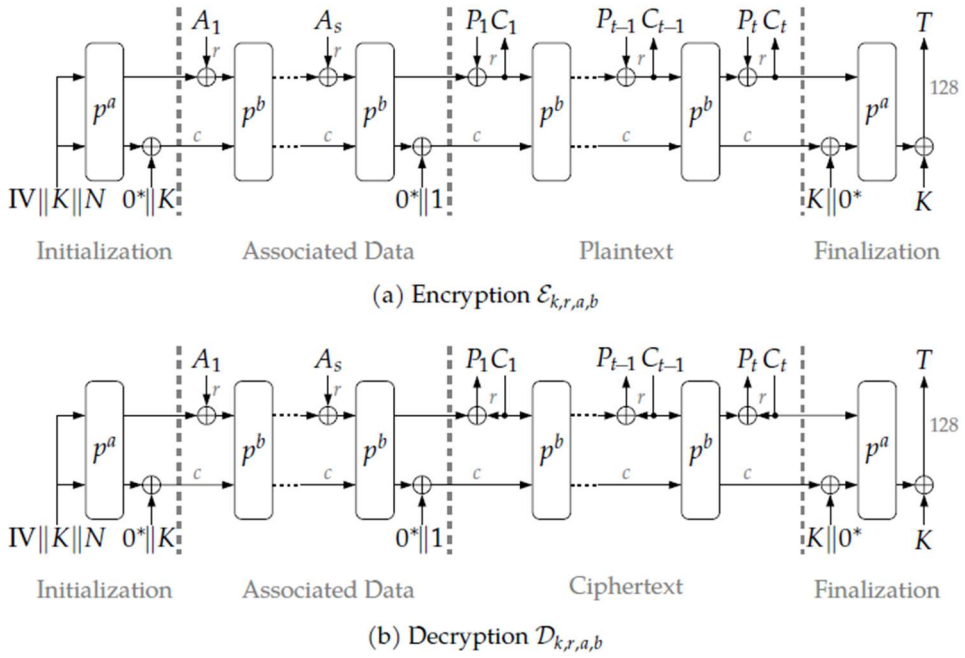


Figure 1.4: ASCON authenticated encryption and decryption algorithms

The hashing algorithm will not be mentioned in the report, because it will not be implemented to Ibex.

1.3.1 Encryption algorithm

ASCON encryption algorithm consists of four stages: Initialization, processing associated data, processing plaintext, and finalization. [3]

Initialization

In this stage, the initialization vector (IV) and 320-bit state of the sponge construction (S) are created. IV is created with the concatenation of k, r, a, b and zeroes:

$$IV \leftarrow k \,||\, r \,||\, a \,||\, b \,||\, 0^{(160-k)} \quad [3]$$

S is created by concatenating the key (K) and the nonce (N) with IV:

$$S \leftarrow IV \,||\, K \,||\, N \quad [3]$$

Finally, a rounds of the round transformation p are applied to the initial state (S), S is XORed with the key (K) and the result is assigned to S:

$$S \leftarrow p^a(S) \oplus (0^{320-k} || \, K) \quad [3]$$

Processing of associated data

Firstly, padding is applied to associated data by separating it into blocks of r bits. A single 1 and zeroes are appended to data to obtain a multiple of r bits, and the data are split to s blocks of r bits:

$$A_1, \ldots, A_s \leftarrow r - bitblocksofA \,||\, 1 \,||\, 0^{r-1-(|A| mod r)} \quad [3]$$

Each block of A_i (i = 1, …, s), are XORed with the most significant r-bits of state (S), and are applied b rounds of round permutation p^b:

$$S \leftarrow p^b\left((S_r \oplus A_i)|| S_c\right) \quad [3]$$

Also, padding is optional. If associated data is zero, nothing will be appended to the data and the data will not be split.

Finally, state (S) is XORed with a single 1 and 319 zeros, and assigned to the S:

$$S \leftarrow S \oplus (0^{319}|| 1) \quad [3]$$

Processing plaintext

Firstly, padding is applied to plaintext in the same steps as did while processing associated data and the result is split t blocks of r bits:

$$P_1, \ldots, P_t \leftarrow r - bitblocksofP|| 1 \,||\, 0^{r-1-(|P| mod r)} \quad [3]$$

In every iteration, one P_i block (i = 1, .., t) is XORed with the most significant r bits of state (S_r) and the result is assigned to ciphertext block C_i. After that, the least

significant c bits of S (S_c) are appended to C_i, and b rounds of round permutation are applied to them. The result is assigned to S:

$$C_i \leftarrow S_r \oplus P_i \quad [3]$$

$$S \leftarrow p^b(C_i \,||\, S_c) \quad [3]$$

However, in the last iteration, after appending C_i to S_c, the result will be assigned to S without applying the round permutation:

$$S \leftarrow C_i||S_c \quad [3]$$

Finally, the last ciphertext block C_t shortened to the length of the unpadded last plaintext block by eliminating its the most significant $|C_t|$-r+1 bits:

$$C_t \leftarrow C_t(r-1)C_t(r-2)\dots C_1 \quad [3]$$

Finalization

In the last stage, the key (K) is XORed with the state (S) and *a* rounds of round permutation is applied to the result:

$$S \leftarrow p^a\Big(S \oplus (0^r \,||\, K \,||\, 0^{c-k})\Big) \quad [3]$$

The tag T is the result of XORing the least significant 128 bits of the state (S) and the last 128 bits of the key (K):

$$T \leftarrow S^{128} \oplus K^{128} \quad [3]$$

1.3.2 Decryption algorithm

The decryption algorithm consists of four stages: Initialization, processing of associated data, processing ciphertext, and finalization. The first two stages are the same as the first two stages in the encryption, but the last two steps are different.

1.3.2.1 Processing plaintext

Firstly, the ciphertext is split t blocks of r bits:

$$C_1, \ldots, C_t \leftarrow r - bit\ blocks\ of\ C \quad [3]$$

In every iteration, one C_i block (i = 1, .., t) is XORed with the most significant r bits of state (S_r) and the result is assigned to ciphertext block P_i. After that, the least significant c bits of S (S_c) appended to C_i and b rounds of round permutation are applied to them. The result is assigned to S:

$$P_i \leftarrow S_r \oplus C_i \quad [3]$$

$$S \leftarrow p^b(C_i || S_c) \quad [3]$$

In the last iteration, shortened the last ciphertext block with l ($0 \leq l <$ r) is XORed with the most significant r bits of S_r, the result assigned to P_t and state (S) is updated:

$$P_t \leftarrow {S_r}^l \oplus C_t \quad [3]$$

$$S \leftarrow \left(S_r \ \oplus (P_t || \ 1 \ || \ 0^{r-1-l})\right) || \ S_c \quad [3]$$

1.3.2.2 Finalization

The steps of the final stage are the same as encryption. However, plaintext is only produced when the received tag value is equal to the produced tag value.

1.3.3 Permutation

The round permutation is a function that is used in the algorithms multiple times. It gets 320-bit state (S) as input parameter and splits five equal 64-bit registers:

$$S = x_0 || \ x_1 || \ x_2 || \ x_3 \ || \ x_4 \quad [3]$$

It consists of three stages: Addition of constants, substitution layer, linear diffusion layer.

1.3.2.1 Addition of constants

In this stage, a constant value is XORed with the x_2 register and assigned to the x_2. The constant value gets different values in different a and b values. [3] The values can be seen in Figure 1.5.

$$x_2 \leftarrow x_2 \oplus c_r \quad [3]$$

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			0000000000000000f0	6	2	0	0000000000000000096
1			0000000000000000e1	7	3	1	0000000000000000087
2			0000000000000000d2	8	4	2	0000000000000000078
3			0000000000000000c3	9	5	3	0000000000000000069
4	0		0000000000000000b4	10	6	4	000000000000000005a
5	1		0000000000000000a5	11	7	5	000000000000000004b

Figure 1.5: Constant values. [3]

1.3.2.2 Substitution layer

In this stage, state (S) splits into 64 pieces that are 5 bits long, and the pieces are updated with the s-box function [3]. The slicing of the five registers can be seen in Figure 1.6.

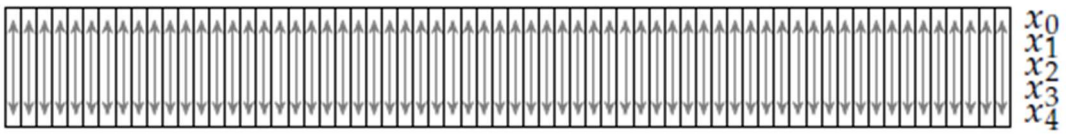


Figure 1.6: Slicing of the five registers in the substitution layer. [3]

S-box function can be seen in Table 3. However, implementing the layer in processors can be hard due to parallel operations. An alternative way to implement the layer can be seen in Figure 1.8.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$\mathcal{S}(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Figure 1.7: Inputs and outputs of S-box function [3]

```
x0 ^= x4;    x4 ^= x3;    x2 ^= x1;
t0  = x0;    t1  = x1;    t2  = x2;    t3  = x3;    t4  = x4;
t0 =~ t0;    t1 =~ t1;    t2 =~ t2;    t3 =~ t3;    t4 =~ t4;
t0 &= x1;    t1 &= x2;    t2 &= x3;    t3 &= x4;    t4 &= x0;
x0 ^= t1;    x1 ^= t2;    x2 ^= t3;    x3 ^= t4;    x4 ^= t0;
x1 ^= x0;    x0 ^= x4;    x3 ^= x2;    x2 =~ x2;
```

Figure 1.8: Alternative way to implement substitution layer [3]

1.3.2.3 Linear diffusion layer

The linear diffusion layer updates every register x_i differently. Operations of the layers can be seen in Figure 1.9.

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

Figure 1.9: Operations of linear diffusion layer [3]

2. SETTING UP AND TESTING THE TOOLS

2.1 RISC-V GNU Compiler Installation

To run the ASCON algorithm on the Ibex, the C code of the algorithm had to be compiled and the memory file that contains instructions machine code had to be created. Therefore, firstly the RISC-V GNU Compiler was installed.

For installation, it is necessary to download some packages to the computer first and if not, git must be installed for download from Github. After doing this, the RISC-V GNU Toolchain is installed by following the commands below:

- git clone https://github.com/riscv/riscv-gnu-toolchain
- sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
- ./configure --prefix=/opt/riscv
- make

By default, the build targets the RV64GC. To create the RV32GC toolchain, use the commands below:

- ./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=ilp32d
- make linux

The 32-bit RISC-V GNU Toolchain is installed, thus moving on to the next stage and a sample C program is run and the Compiler and Ibex are tested for it is working correctly.

2.2 Running A Sample Program

Before trying to run the ASCON algorithm, we wanted to make sure everything was working correctly by compiling a simple C code and running it on Ibex. For this, a code in the form of a + b = c was written, and it was run on Ibex by making necessary operations.

```c
#include <stdint.h>

int main() {

int a = 4545;
int b = 751;

int *c;
*c = a + b;

return 0;
}
```

Figure 2.1: Simple C Program

To create the memory file and other necessary files, Makefile should be used by typing the following basic command in the terminal:

- make

To make a new compilation process, it is necessary to delete the previously created and unnecessary files. For this, the following command should be used:

- make distclean

From the simulation results, the result of the summation was seen as correct and thus it was concluded that the conditions were ready to move on to the next stage.

Figure 2.2: Simulation Results of Simple C Program

3. RUNNING AND PROFILING ASCON ALGORITHMS ON IBEX

3.1 Running ASCON Encryption Algorithm

After running a sample program on Ibex, software implementation of the ASCON encryption algorithm is run on Ibex. The source files can be found in the official Github repository of ASCON. Main.c, round.c and encrypt.c compiled in RV32I instruction set by using RISC-V GNU Toolchain to run on Ibex.

The software implementation is done according to recommended ASCON128a parameters. The parameters can be seen in Figure 3.1.

Name	Algorithms	Bit size of				Rounds	
		key	nonce	tag	data block	p^a	p^b
Ascon-128	$\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6
Ascon-128a	$\mathcal{E}, \mathcal{D}_{128,128,12,8}$	128	128	128	128	12	8

Figure 3.1: Recommended parameters for the implementation of ASCON [3]

Input parameters of the encryption function can be seen in Figure 3.2.

```
unsigned char a[] = "thisistheassociateddata.";
unsigned char m[] = "thisisaplaintext";
unsigned char c[16 + CRYPTO_ABYTES];
unsigned char nsec[CRYPTO_NSECBYTES];
unsigned char npub[CRYPTO_NPUBBYTES] = {"thisisthenonce."};
unsigned char k[CRYPTO_KEYBYTES] = "thisisthekey.";
```

Figure 3.2: Input parameters of the encryption function

Outputs of the encryption function, ciphertext and tag can be seen in Figure 3.3. The outputs are written in hexadecimal radix to identify in simulation easily.

```
Cipher Text
c[16]=c9, 2c, e2, f4, 6a, 2c, 62, 1d, 82, 97, 4f, 9f, d0, 85, eb, 38,

Tag
t[16]=55, 9, 32, 3d, ff, df, 15, 17, ea, 2e, bb, 35, b, 23, ca, 2b,
```

Figure 3.3: Outputs of the encryption function

The simulation results can be seen in Appendicies. Figure A.1a and A.1b show ciphertext which was written to the memory, Figure A.1c and A.1d show the tag which was written to the memory. LED1 signal shows the data which is written to

memory or read from memory. The tag and the ciphertext are produced and written to memory successfully in 66952 clock cycles.

3.2 Running ASCON Decryption Algorithm

After running an encryption algorithm on Ibex, software implementation of ASCON decryption algorithm is run on Ibex. The compilation was done similarly to encryption and main.c and decrypt.c compiled to run on Ibex. Their codes can be found in Appendices.

Output of the decryption function, plaintext can be seen in Figure 3.4. The output is written in hexadecimal radix to identify in simulation easily.

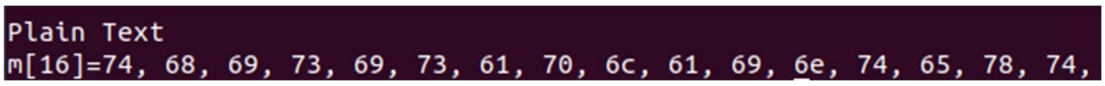


Figure 3.4: Output of the decryption function

Figure B.1a and b shows that the plaintext were produced and written to memory successfully in 66198 clock cycles.

3.3 Deciding the Most Frequent Function

C code of the encryption algorithm is profiled by using SPIKE RISC-V ISA Simulator and RV32I instruction set. The decryption algorithm did not profile because it is mostly the same as the encryption algorithm.

Five optimization flags are used in profiling and they are named -O0, -O1, -O2, -O3, and -Os. The compiler makes no optimization on the code with the -O0 flag. -O1 flag makes optimization on code size and execution time for small functions. -O2 makes all supported optimization for the code but does not change the space-speed balance. -O3 does the same optimizations as -O2 and does more optimizations than it. -Os optimizes to only reduce the code size. [8]

Profiling results about the round permutation block which is the most used can be seen in Figure 3.5. Figure 3.5 states that the linear layer and the substitution layer are the most time-consuming functions. [8]

Optimization	Round constant addition	Substitution layer	Linear layer
-Os	2,5%	45,2%	52,3%
-O3	2,5%	45,2%	52,3%
-O2	2,5%	45,2%	52,3%
-O1	2,5%	45,2%	52,3%
-O0	3,6%	48,7%	47,7%

Figure 3.5: Profiling results of the round permutation in all optimization flags. [8]

So, s-box and round functions must be designed as a custom module for the ALU.

3.3.1 Round function

One of the most frequent functions is the ROTR32 function. It gets two parameters: 32-bit shifting amount and 32-bit number to shift. It shifts all bits of the number right as shifting amount and fills vacant bits with the least significant bits. In the testbench, 3 different numbers shifted 3 different amounts. Simulation results can be seen in Figure 3.6.

Figure 3.6: Simulation results of the testbench.

RTL schematic of the module, the Verilog implementation and testbench for the implementation can be found in Appendix.

3.3.2 S-box function

The other most frequent function is the s-box function. The custom module for the S-box function can be seen in Figure 3.7.

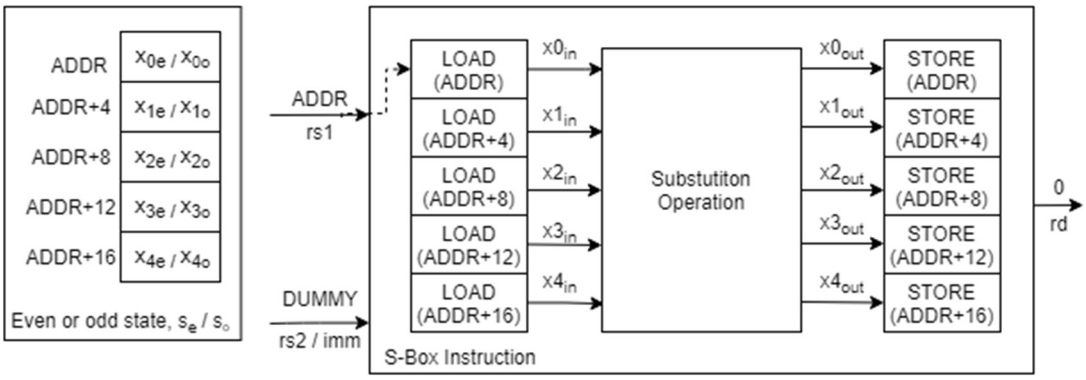


Figure 3.7: The custom module for the s-box [8]

Two arrays are named even and odd states in the software implementation of the ASCON algorithm. The custom module takes one of them as the first operand. The second operand is not used and the output is always zero. The module takes the address of the first element of the input array and holds it. After that, it loads all elements of the array to the module by using the direct memory access to the block RAM and makes the s-box computations that are shown in Figure 1.8. Finally, it loads the outputs of the function to the input array. Verilog implementation of s-box module is given in Appendicies.

4. INSTRUCTION SET EXTENSION OF IBEX

4.1 Changes in Software

While adding the instructions, the compiler is modified to detect them. The functions are named cust0 and cust2 in the compiler for simplicity.

There are two files named risc-v-opc.c and risc-v-opc.h that contain instruction templates in the riscv-gnu-compiler folder. Risc-v-opc.c and risc-v-opc.h files must be modified to add instructions. risc-v-opc.c file contains the list of opcodes and their properties. The template and properties of new instructions will be written into the riscv_opcodes array in the file.

risc-v-opc.h contains definitions for risc-v-opc.c file. The MATCH and MASK values are defined and associated with the instruction in the file. Association of MATCH and MASK values with the instruction is done with the DECLARE_INS function. After the modifications, generated compiler files must be deleted and the compiler must be reinstalled. The template of cust0 instruction can be seen in Figure 4.1.

```
//Added instructions by Yunus Emre ERYILMAZ
{"cust0",   0, INSN_CLASS_I, "d,s,t", MATCH_CUST0, MASK_CUST0, match_opcode, 0},
```

Figure 4.1: The cust1 instruction template in risc-v-opc.c

There are 8 arguments to define instructions in the compiler. The first one is the instruction name. The second one is the requirement of XLEN (width of the integer register size in bits) for the instruction, cust0 is 32-bit, so it is declared as zero. The third one is the class of the instruction, it can be I, M, or C. Cust0 is an instruction in the I class. The fourth one describes the arguments of the instruction, the new instruction is R-type instructions, so their operands are *"d,s,t"* in which *d* is for destination register, and *s* and *t* are for source operands. The fifth, sixth and seventh arguments are for decoding the instruction. If ANDing of the incoming instruction word and the MASK_CUST0 is matched with MATCH_CUST0, the compiler will recognize the instruction as cust0. So, the MATCH value must be unique to differentiate from the other instructions.

risc-v-opc.h contains definitions for risc-v-opc.c file. The MATCH and MASK values are defined and associated with the instruction in the file. Association of

MATCH and MASK values with the instruction is done with the DECLARE_INSN function. Added lines in risc-v-opc.h can be seen in Figure 4.2 and 4.3.

```
#define MATCH_CUST0 0x30000033
#define MASK_CUST0 0xfe00707f

#define MATCH_CUST1 0x50000033
#define MASK_CUST1 0xfe00707f

#define MATCH_CUST2 0x0600700B
#define MASK_CUST2 0xfe00707f
```

Figure 4.2: MATCH and MASK values for custom instruction.

```
DECLARE_INSN(cust0, MATCH_CUST0, MASK_CUST0);
DECLARE_INSN(cust1, MATCH_CUST1, MASK_CUST1);
DECLARE_INSN(cust2, MATCH_CUST2, MASK_CUST2);
```

Figure 4.3: Added DECLARE_INS lines.

After the modifications, generated compiler files must be deleted, and the compiler must be reinstalled:

- sudo make distclean
- ./configure --prefix=/opt/riscv --with-arch=rv32imc
- sudo make

Also, inline assembly statements are added to the ASCON source files to call added custom instructions. Inline assembly is a method to call low-level language statements in a script that is written in a high-level language [9]. Using inline assembly increases optimization, decreases function-call overhead, and allows to use of CPU-specific instructions. However, inline assembly makes the maintenance and the readability of the code harder. The code with inline assembly cannot be cross-platform because the used instructions can be CPU-specific, and they cannot run on the other platforms. Any compiler can handle the inline assembly differently and this can cause a decrease in performance.

The inline assembly statement to call cust2 is shown in Figure 4.4. *asm* means "assembly" and *volatile* indicates that optimization will not be applied to this statement. "*"cust2 %[result0], %[value1], %[value2]\n\t": [result0] "=r" (r0) : [value1] "r" (&s_e[0]) , [value2] "r" (1)*" is the assembly code to call cust2. *result0*

is the placeholder for the destination register, *value1* and *value2* are placeholders for the operands. *r* indicates the operand is a register, = in *=r* indicates the result will be overwritten to an existing value. The contents inside of the parentheses are operands of the instruction.

```
asm volatile("cust2 %[result0], %[value1], %[value2]\n\t":
  [result0] "=r" (r0) : [value1] "r" (&s_e[0]) , [value2] "r" (1)
);
```

Figure 4.4: The inline assembly statement to call cust2.

4.2 Changes in Hardware

Modifications in processor architecture are done in the decoder and the ALU while adding the round module. The round function is a single cycle operation, so it can be instantiated in parallel to the other ALU operations as AND, OR, and XOR. The decoder must be modified to recognize the round instruction. So, a new case was added to the case structure in the OPCODE_OP case. The new case is shown in Figure 4.5.

```
unique case ({instr[30:25], instr[14:12]})
  // RV32I ALU operations
  {6'b00_0000, 3'b000}: alu_operator_o = ALU_ADD;   // Add
  {6'b10_0000, 3'b000}: alu_operator_o = ALU_SUB;   // Sub
  {6'b00_0000, 3'b010}: alu_operator_o = ALU_SLT;   // Set Lower Than
  {6'b00_0000, 3'b011}: alu_operator_o = ALU_SLTU;  // Set Lower Than Unsigned
  {6'b00_0000, 3'b100}: alu_operator_o = ALU_XOR;   // Xor
  {6'b00_0000, 3'b110}: alu_operator_o = ALU_OR;    // Or
  {6'b00_0000, 3'b111}: alu_operator_o = ALU_AND;   // And
  {6'b00_0000, 3'b001}: alu_operator_o = ALU_SLL;   // Shift Left Logical
  {6'b00_0000, 3'b101}: alu_operator_o = ALU_SRL;   // Shift Right Logical
  {6'b10_0000, 3'b101}: alu_operator_o = ALU_SRA;   // Shift Right Arithmetic

  {6'b01_1000, 3'b000}: alu_operator_o = ALU_CUST0;
```

Figure 4.5: Modified case structure of ALU operations.

The round function is instantiated in the ALU module. operand_a_i and operand_b_i are the inputs, cust0_result is the output of the custom module. Also, the result mux is modified to conduct the result of the custom module. Modified result mux can be seen in Figure 4.6.

```
////////////////
// Result mux //
////////////////

always_comb begin
  result_o   = '0;

  unique case (operator_i)
    // Standard Operations
    ALU_AND:  result_o = operand_a_i & operand_b_i;
    ALU_OR:   result_o = operand_a_i | operand_b_i;
    ALU_XOR:  result_o = operand_a_i ^ operand_b_i;

    // Adder Operations
    ALU_ADD, ALU_SUB: result_o = adder_result;

    // Shift Operations
    ALU_SLL,
    ALU_SRL, ALU_SRA: result_o = shift_result;

    // Comparison Operations
    ALU_EQ,   ALU_NE,
    ALU_GE,   ALU_GEU,
    ALU_LT,   ALU_LTU,
    ALU_SLT,  ALU_SLTU: result_o = {31'h0,cmp_result};

    ALU_CUST0: result_o = cust0_result;
    default:;
  endcase
end
```

Figure 4.6: Modified result mux.

The s-box module is instantiated in the execution stage, in parallel to the multiplication module because it does a multi-cycle operation. Appropriate input and output signals are created to operate the module in the execution. The inputs and outputs of the stage are changed to control the block RAM and get data from the other modules. The decoder and controller in the instruction decode stage are also modified. A new case for the custom instruction is added to the decoder to enable the custom module and the register file. The new case is shown in Figure 4.7. An output signal is added to the controller to stall the processor while waiting to finish multi-cycle operations.

```
//Custom module operation
OPCODE_CUSTOM_0: begin
    custom_en_o = 1'b1;
    regfile_we  = 1'b1;
end
```

Figure 4.7: Newly added case for opcode case structure.

The core will continue to operate when the operation of the custom module is finished. If the core did not enter a waiting state, it can produce wrong results. The source operand and enable output signals from the ID stage are connected to source operands and enable input signals of the EX stage in the ibex_core module. The RAM data, the RAM address and the enable signals are added as inputs and outputs of the core module to communicate with the RAM. The new core signals are connected to the block RAM in the top module.

5. TIMING AND PERFORMANCE ANALYSIS

After modifying the core, the compiler and the source codes, the encryption and the decryption algorithms are run on the core and the operations are validated by comparing each other. In this section, the implementation of ASCON algorithms with and without custom instruction are compared.

5.1 Area Comparison

The area measurement is made by implementing unmodified and modified Ibex core and comparing the utilization reports of them. Areas covered by unmodified and modified Ibex is shown in Table 5.1. The amount of FFs and LUTs are increased by

1.94% and 18.97% respectively because s-box and ROTR32 modules are added to the processor.

Table 5.1: The area comparison

	Unmodified	Modified	Change
LUT	2977	3542	+18.97%
FF	1955	1993	+1.94

5.2 Instruction Memory Size Comparison

Instruction memory sizes are compared by examining the number of lines in the generated vmem files. The comparison is shown in Table 5.2. The program with custom instruction has a smaller number of lines than the other one since the round function without custom instruction is 216 instructions and the same function with custom instruction is 59 instructions.

Table 5.2: The instruction memory size comparison

	Unmodified	Modified	Change
Lines	4251	4153	-2.3%

5.3 Execution time comparison

Execution times of the programs are measured by measuring the execution of the main function takes how many clock cycles in the behavioural simulation of XSim. The comparison between the two programs is shown in Table 5.3. The program with custom instructions takes less time to complete because the instruction memory size of the program with custom instructions is smaller and the custom module operations take less time than the ALU operations.

Table 5.3: Execution time comparison

	Unmodified	Modified	Change
Clock cycle	133150	92830	-30.2%

5.4 Average energy consumption

SAIF file is needed to measure the average energy consumption in the processor, and it is generated after running a post-implementation timing simulation. SAIF (Switching Activity Interchange Format) contains information about static probability and toggle rates of the nets [10]. Static probability is the fraction of time that the signal is HIGH on the net through the system operation [11]. The power report for the processor is generated by using the SAIF file in Vivado and dynamic power can be learned with power reports. Average energy consumption is the multiplication of the execution time, the clock frequency, and the dynamic power of the processor. The average energy consumption comparison of the program with and without custom instructions is shown in Table 5.4. The program with custom instructions consumes less energy than the program without custom instructions because the execution time is shorter than the other program. So, switching activity and the energy consumption is lesser too.

Table 5.4: Average energy consumption comparison

	Unmodified	Modified	Change
Average dynamic power [W]	0.228	0.196	-14.03%
Minimum clock period [ns]	7.859	7.645	-2,72%
Average energy consumption [mJ]	0.239	0.139	-41.84%

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

Lightweight cryptography is an important field for security at a time when the usage of constrained devices is increasing. ASCON is one of the lightweight cryptography algorithms and is close to being a standard for LWC. It is a fast and efficient algorithm even in its software implementation and it can be more resourceful with instruction set extension. To show it, the C program for ASCON is firstly compiled and run on Ibex core. After observing the whole program running on the core, profiling of the program is made, and the most frequent operations are decided. Then, equivalent modules are designed in Verilog language and instantiated into the core. Finally, the execution time, the average energy consumption, the instruction memory size, and the area usage are measured and compared. According to the results, the area usage increased but the execution time, the average energy consumption, and the instruction memory size is decreased. Therefore, the ISA extension has increased the performance of the algorithm.

6.1 Practical Application of this Project

The project is appliable for fields where security is important and constrained devices are used like healthcare, IoT, and cyber-physical systems.

6.2 Realistic Constraints

The used sources and tools are open-source and free. So, examining, designing, and debugging with the sources makes the projects cost-effective but these sources are new, there is not enough documentation for them. Using them sometimes can be time-consuming.

6.3 Social, environmental, and economic impact

In the 21st century, more information is shared than ever in history. Information security is more important, and leakage of data is more vital. So, every secret must

be encrypted and protected properly. It is of utmost importance to design equipment that will ensure this healthy and secure sharing environment and implement crypto protocols. The fast and efficient design of this equipment will make information sharing safer.

6.4 Cost analysis

Personal computers, open-source tools, open-source designs and Vivado WebPACK are used during the project. So, there is no cost in the project.

6.5 Standards

The modified core itself will be following RISC-V ISA standards and hardware implementation will be following IEEE standards. ASCON algorithm is under development of standardization.

6.6 Health and safety concerns

There are no health and safety concerns about the project because the project is designed and tested on simulation programs.

6.7 Future Work and Recommendations

The aim for future projects can be implementing a communication module for communication with different cores and testing the core easily. Another project can be modifying the GCC for custom functions without the need for inline assembly because using inline assembly decreases the readability of the code and inhibits the optimization of assembly code. Thus, the instruction memory size and the execution time would be decreased.

REFERENCES

[1] K. McKay, L. Bassham, M. Turan Sönmez, and N. Mouha. "Report on lightweight cryptography", National Institute of Standards and Technology, Maryland, Washington, D.C., USA, NIST Internal or Interagency Report (NISTIR) no. 8114, Mar. 2017.

[2] M. Sonmez Turan, K. A. McKay, C. Calik, D. H. Chang and L. E. Bassham, "Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process", National Institute of Standards and Technology, Maryland, Washington, D.C., USA, NIST Interagency/Internal Report (NISTIR) no. 8268, Oct. 7, 2019.

[3] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2 Submission to NIST", 2019. Accessed: Jun 14. 2022. [Online]. Available: https://ascon.iaik.tugraz.at/files/asconv12-nist.pdf

[4] A.S. Waterman (2016). "Design of the RISC-V instruction set architecture" Ph.D. dissertation, Dept. Elect. Eng. and Comp. Sci. Univ. of California, Berkeley, CA, USA, 2016.

[5] A. Waterman, Y. Lee, D.A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", 2019. Accessed: Jun 10. 2022. [Online]. Available: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

[6] A. Waterman, Y. Lee, D.A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.1", 2016. Accessed: Jun 9. 2022. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf

[7] Ibex Documentation, lowRISC, Jun. 14 2022, https://ibexcore.readthedocs.io/

/downloads/en/latest/pdf/.

[8] Ö. Altınay and B. Örs, "Instruction Extension of RV32I and GCC Back End for Ascon Lightweight Cryptography Algorithm," *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS),* 2021, pp. 1-6, doi: 10.1109/COINS51742.2021.9524190.

[9] Gcc.gnu.org. 2022. *DontUseInlineAsm - GCC Wiki*. [online] Available at: <https://gcc.gnu.org/wiki/DontUseInlineAsm> [Accessed 14 January 2022].

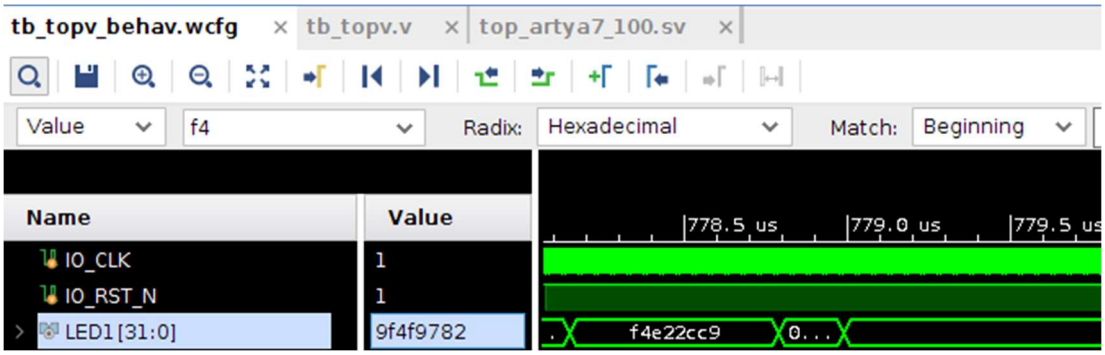
[10] Synopsys.com. 2022. *Power Profile from RTL to Gate-level Implementation |IP| Synopsys*. [online] Available at: <https://www.synopsys.com/designware-ip/technical-bulletin/understanding-power-profile.html> [Accessed 10 January 2022].

[11] *Intel® Quartus® Prime Pro Edition User Guide: Power Analysis and Optimization* (2021). Accessed: Jan. 11, 2022. [Online]. Available: https://d2pgu9s4sfmw1s.cloudfront.net/DITA-technical-publications/PROD/PSG/ug-qpp-power-683174-709286.pdf

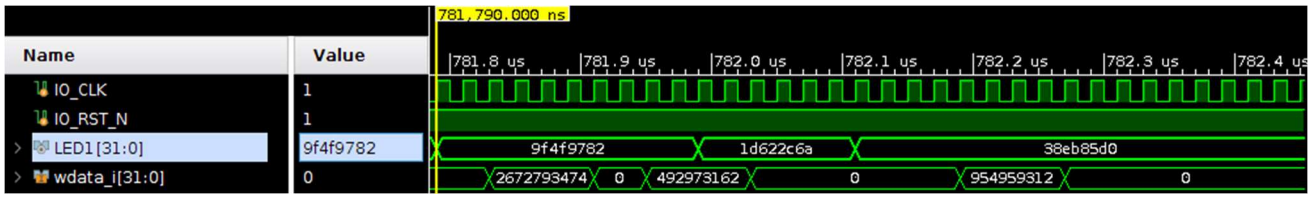
APPENDICES

APPENDIX A: Simulation results of the encryption algorithm that run on Ibex
APPENDIX B: Simulation results of the decryption algorithm that run on Ibex
APPENDIX C: Verilog implementation of ROTR32 function
APPENDIX D: Testbench for Verilog implementation of ROTR32 function
APPENDIX E: Verilog implementation of s-box operation
APPENDIX F: RTL schematics of the modified processor

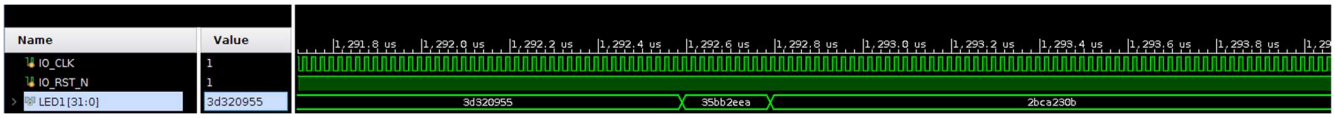
APPENDIX A



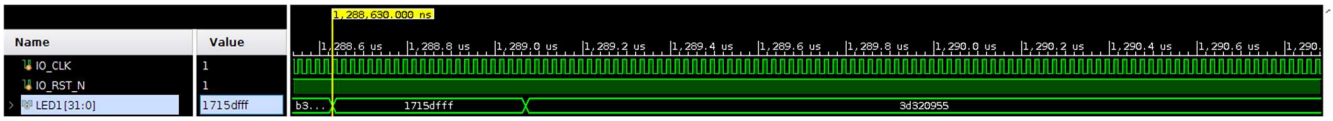
(a)



(b)



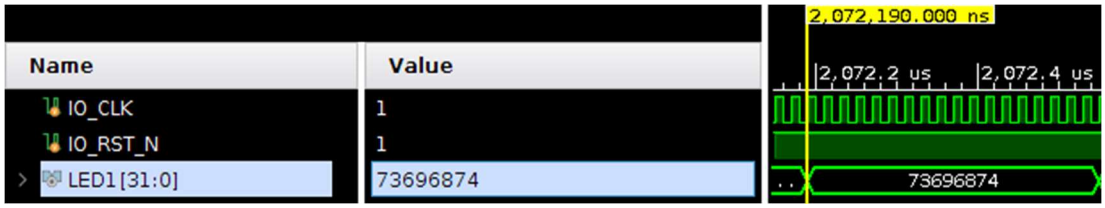
(c)



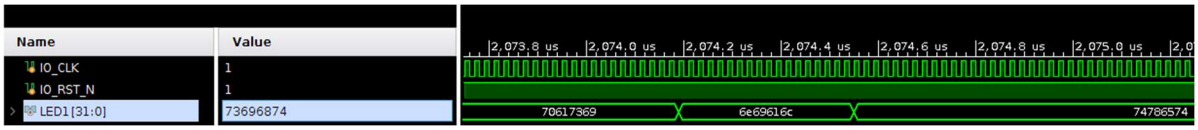
(d)

Figure A.1: Simulation results of the encryption algorithm that run on Ibex

APPENDIX B



(a)



(b)

Figure B.1: Simulation results of the decryption algorithm that run on Ibex

APPENDIX C

```
module ROTR32(

input [31:0] x, n,

output [31:0] result

 );

    assign result = (((x) >> (n)) | ((x) << (32 - (n))));

endmodule
```

APPENDIX D

```
module tb_rotr32;


reg [31:0] x,n;
wire [31:0] result;
ROTR32 dut (x,n,result);


initial begin
    x = 32'hcf655448;
    n = 32'h4;
    #10;
    x = 32'hfbc6a7d2;
    n = 32'h3;
    #10;
    x = 32'he2e6add7;
    n = 32'h8;
    #10;
end
endmodule
```

APPENDIX E

```
module sboxExtension(

    input logic enable, clock,

    input logic [31:0] op_a_i, op_b_i, ram_data_i,

    output logic write_enable,  done,

    output logic [31:0] ram_addr_o, ram_data_o, result

    );



    logic [31:0] sbox_in [5:0];

    logic [31:0] sbox_out [4:0];

    logic [3:0] state;

    logic [3:0] nextState;

    logic [31:0] address = 0;

    integer count = 0;



    sbox_primitive sp01 (.x0(sbox_in[1]),.x1(sbox_in[2]),
.x2(sbox_in[3]),.x3(sbox_in[4]),.x4(sbox_in[5]),.x0_o(sbox_out
```

```
[0]),.x1_o(sbox_out[1]),.x2_o(sbox_out[2]),.x3_o(sbox_out[3]),
.x4_o(sbox_out[4]) );



    always @(posedge clock) begin

        state <= nextState;

        case (state)

            1,3: begin

                count <= count + 1;

            end

            default: begin

                count <= 0;

            end

        endcase

    end



    always @(*) begin

        case (state)

            0: begin

                address <= op_a_i;
```

```
    write_enable <= 0;
    ram_addr_o <= 0;
    ram_data_o <= 0;
    result <= 0;
    done <= 0;

    if (enable)
        nextState <= 1;
    else
        nextState <= 0;
end

1: begin
   write_enable <= 0;
   ram_data_o <= 0;
   result <= 0;
   done <= 0;
```

```
    if (count < 2) begin
     ram_addr_o <= 0;
     nextState <= 1;
    end
    else if (count < 8) begin
     ram_addr_o <= address + (count-2)*4;
     sbox_in[count-2] <= ram_data_i;
     nextState <= 1;
    end
    else begin
     nextState <= 2;
    end
end

2:begin
    write_enable <= 0;
    ram_addr_o <= 0;
```

```
    ram_data_o <= 0;
    result <= 0;
    done <= 0;
    nextState <= 3;
end

3: begin

    write_enable <= 1;
    result <= 0;
    if (count < 5) begin
     ram_addr_o <= address + count*4;
     ram_data_o <= sbox_out[count];
     nextState <= 3;
    end
    else begin
     done <= 1;
     nextState <= 4;
```

```
    end
end

4: begin
     write_enable <= 0;
     ram_addr_o <= 0;
     ram_data_o <= 0;
     result <= 0;
     nextState <= 5;
end

default: begin
    done <= 0;
    write_enable <= 0;
    ram_addr_o <= 0;
    ram_data_o <= 0;
    result <= 0;
    nextState <= 0;
```

```
            end
        endcase
    end

endmodule

module sbox_primitive
(
    input logic [31:0] x0,x1,x2,x3,x4,
    output logic [31:0] x0_o,x1_o, x2_o, x3_o, x4_o
);

logic [31:0] t0, t1, t2, t3, t4;
logic [31:0] x0_temp, x1_temp, x2_temp, x3_temp, x4_temp;

always @(*) begin
    x0_temp = x0;
    x1_temp = x1;
```

```
x2_temp = x2;

x3_temp = x3;

x4_temp = x4;



x0_temp ^= x4_temp;

x4_temp ^= x3_temp;

x2_temp ^= x1_temp;



t0 = x0_temp;

t1 = x1_temp;

t2 = x2_temp;

t3 = x3_temp;

t4 = x4_temp;



x0_temp = t0 ^ (~t1 & t2);

x2_temp = t2 ^ (~t3 & t4);

x4_temp = t4 ^ (~t0 & t1);

x1_temp = t1 ^ (~t2 & t3);
```

```
        x3_temp = t3 ^ (~t4 & t0);



        x1_temp ^= x0_temp;

        x3_temp ^= x2_temp;

        x0_temp ^= x4_temp;



        x0_o = x0_temp;

        x1_o = x1_temp;

        x2_o = x2_temp;

        x3_o = x3_temp;

        x4_o = x4_temp;



    end //Always end


endmodule
```

APPENDIX F

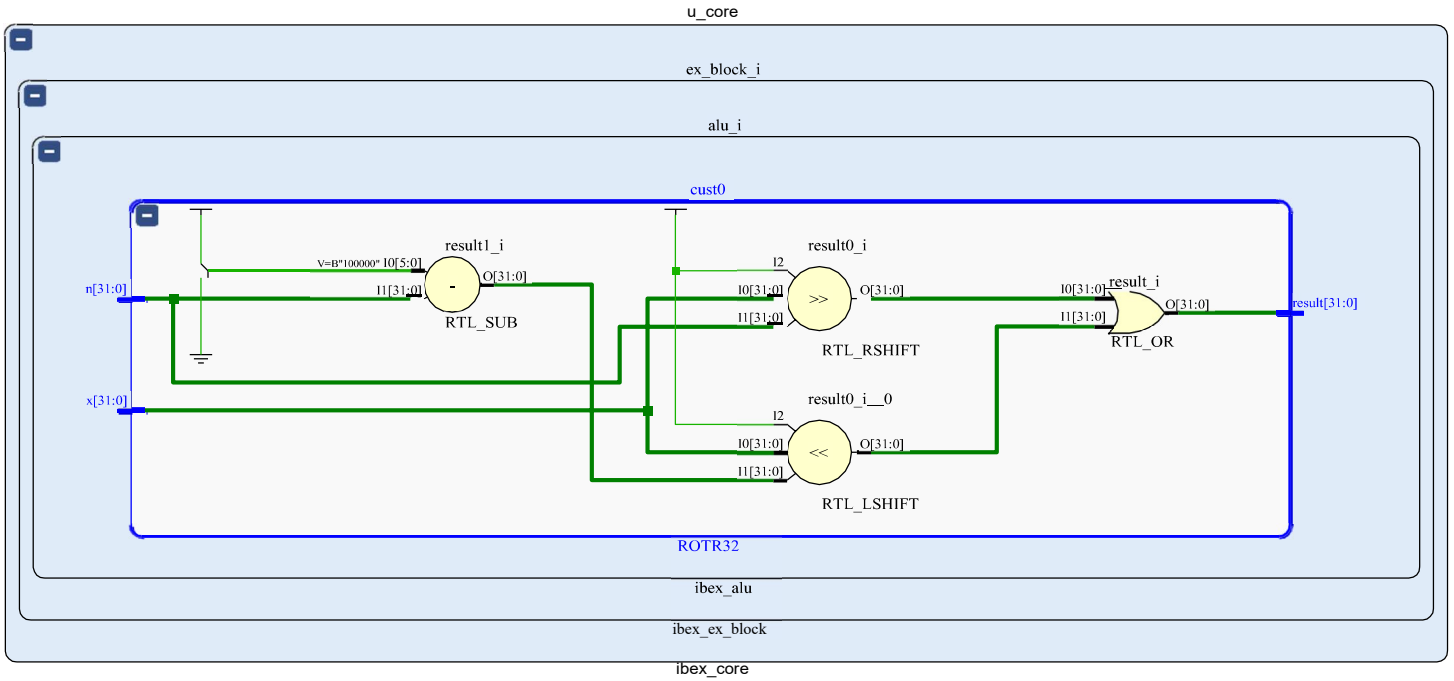


Figure F.1: RTL schematic of the round module.

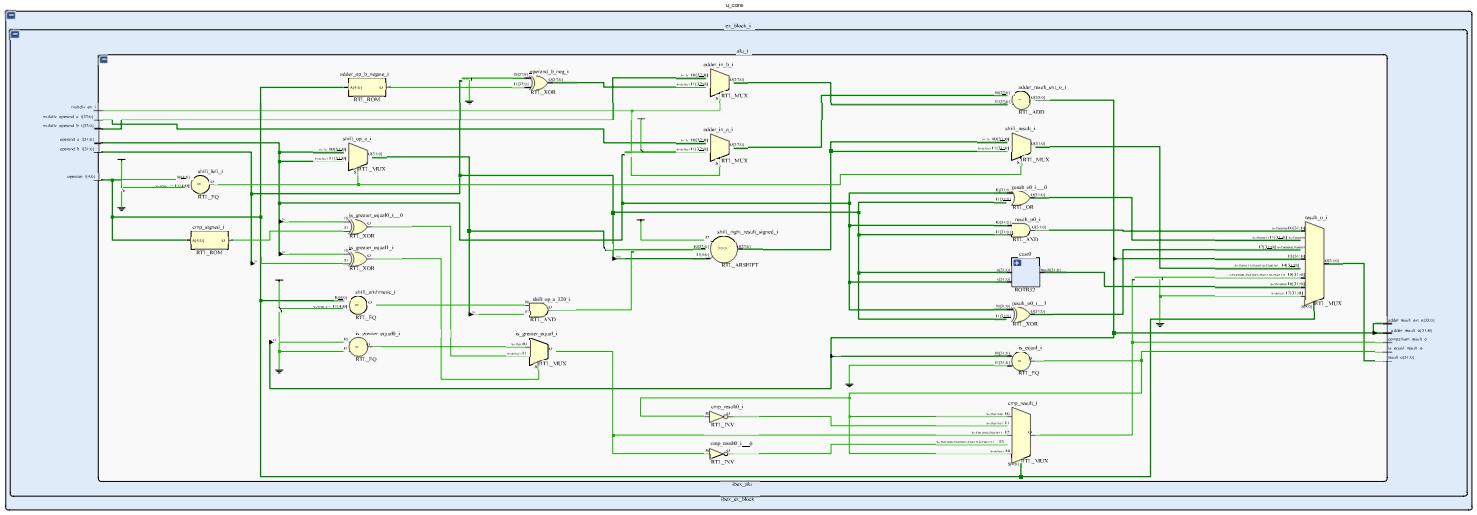


Figure F.2: RTL schematic of the s-box module.

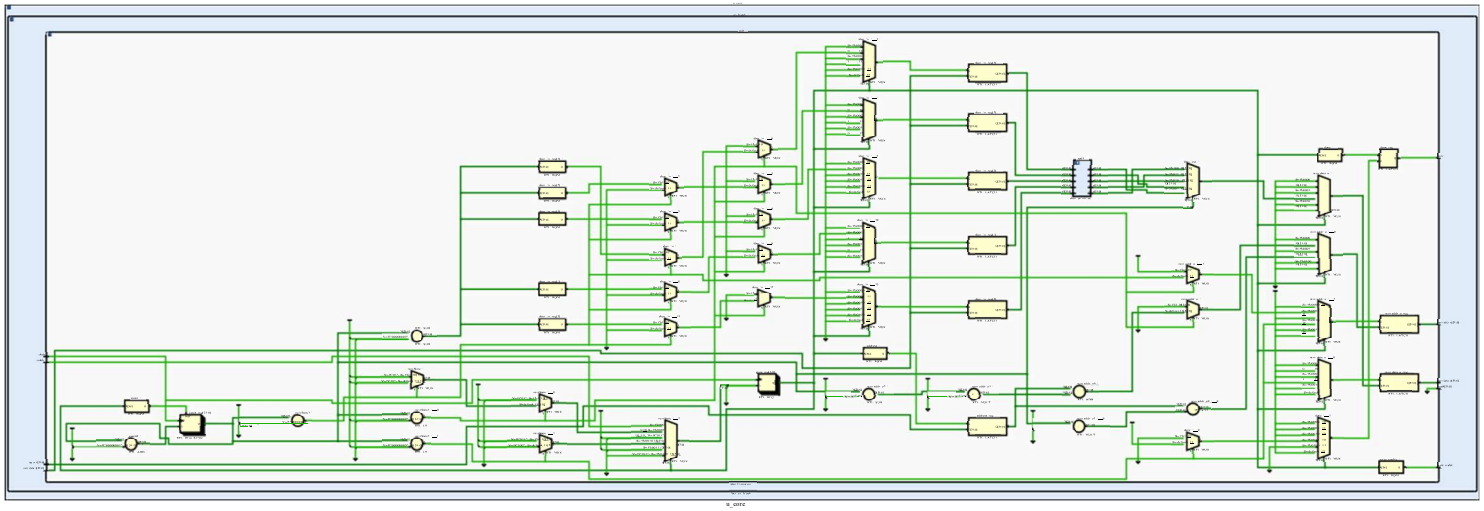


Figure F.3: RTL schematic of the ALU.

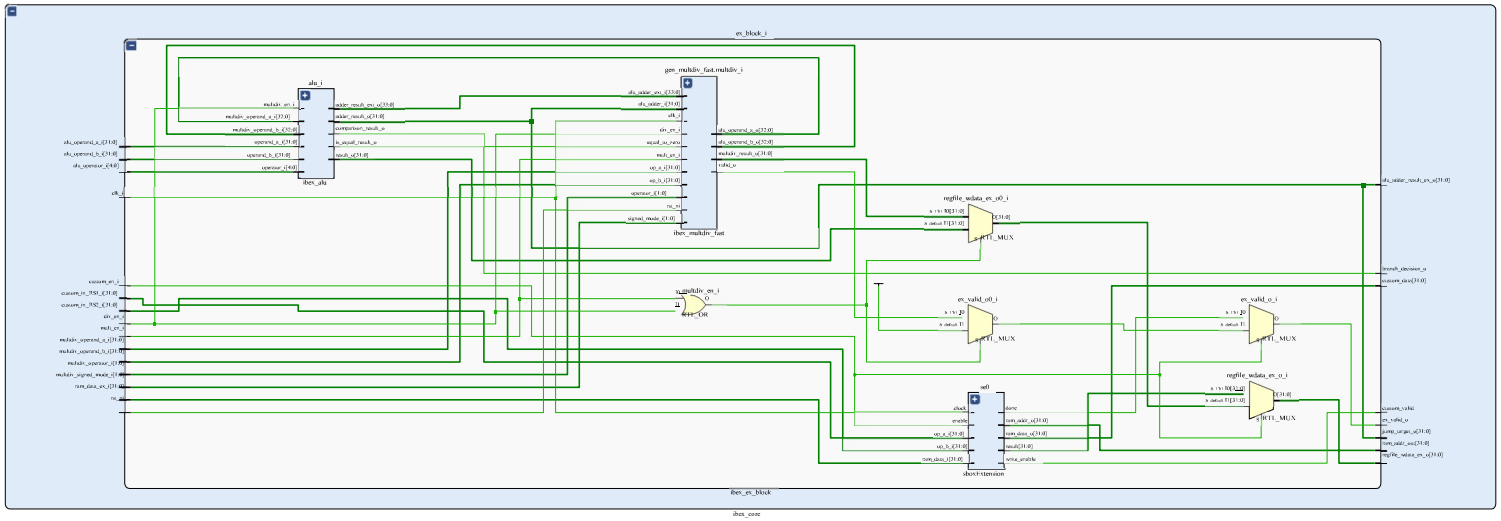


Figure F.4: RTL schematic of the EX block.

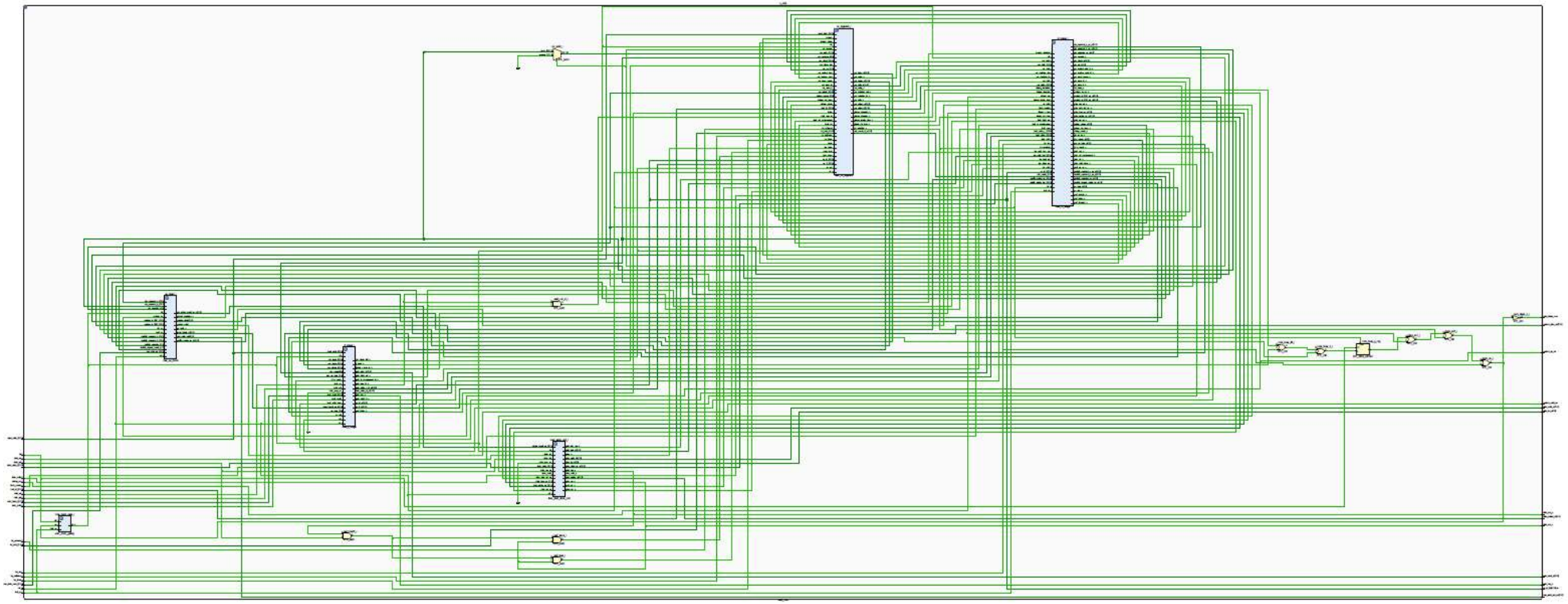


Figure F.5: RTL schematic of the core.

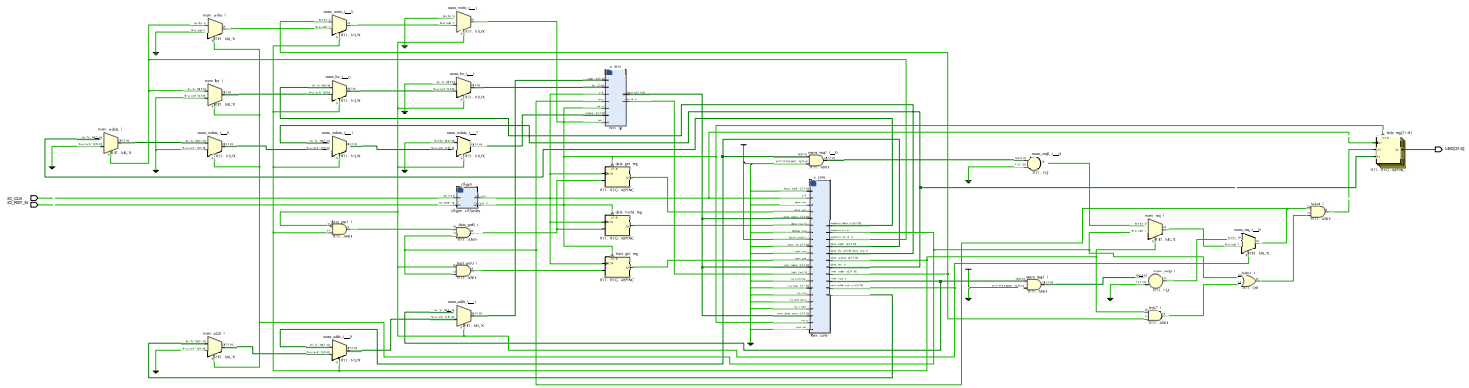


Figure F.6: RTL schematic of the top module.

CURRICULUM VITAE

Name Surname **:** Yunus Emre ERYILMAZ

Place and Date of Birth **:** Kadıköy / 18.09.1999

E-Mail **:** eryilmazy18@itu.edu.tr

Yunus Emre ERYILMAZ is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University. His primary research areas are digital design, computer architecture, cryptology, and embedded systems. He completed his internships at Penta Elektronik A.Ş, TÜBİTAK BİLGEM, and PAVOTEK. He also completed his long-term internship at TÜBİTAK BİLGEM.